COMP20290 Algorithms Assignment

Huffman Compression

This assignment was completed by:

Ahmed Jouda	**18329393**
Ravikanth Gollapalli	**18361993**

This PDF contains:

- Task 1: A hand developed Huffman tree.
- Task 2: Where to find the code.
- Task 3: An Analysis of the huffman algorithm with various inputs.

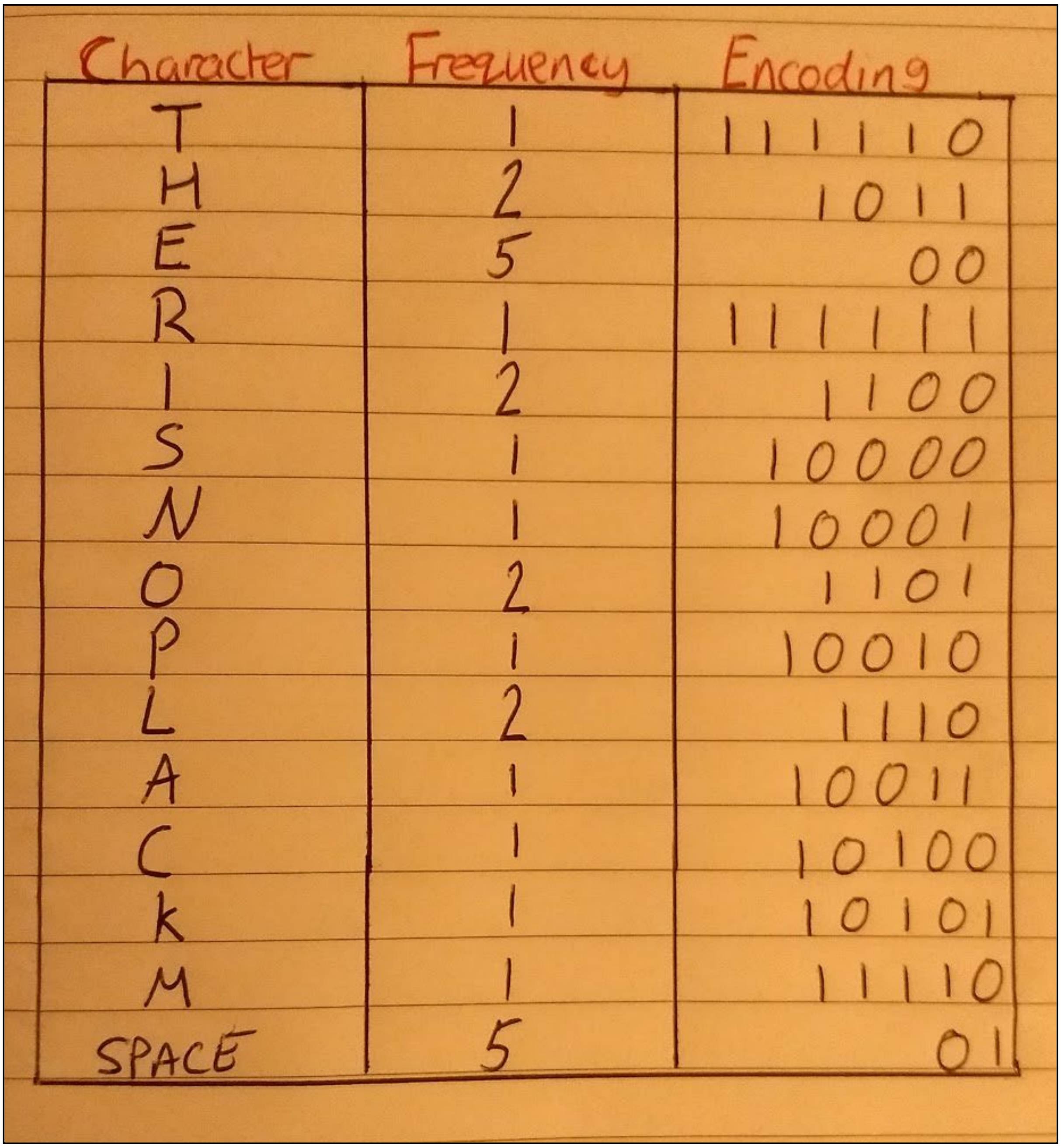
Links to our Github classroom repositories.

Ahmed's Repository:
https://github.com/CompAlgorithms/algorithm-portfolio-20290-AhmedJouda2000.git

Ravikanth's Repository:
https://github.com/CompAlgorithms/algorithm-portfolio-20290-ravig00.git

Task 1: Code Huffman Tree of Phrase by Hand

The first task required us to create a huffman tree from the phrase *"There is no place like home"* by hand.
To do this, I first created a table which includes the characters, the frequency and the encoding:
I counted the characters in the input phrase (including the space character) and noted the amount of times that character appeared in the input phrase.

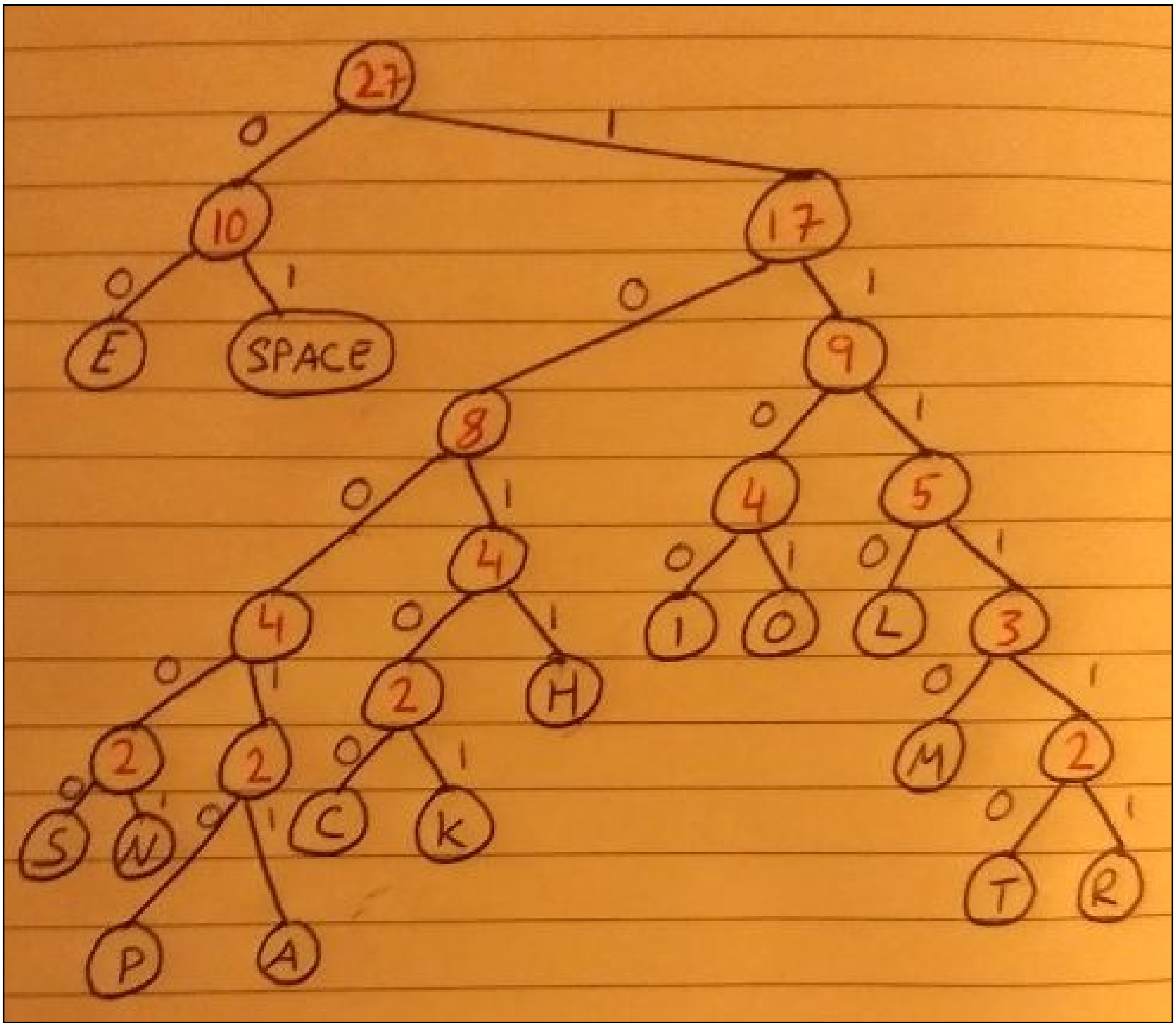


Character	Frequency	Encoding
T	1	111110
H	2	1011
E	5	00
R	1	111111
I	2	1100
S	1	10000
N	1	10001
O	2	1101
P	1	10010
L	2	1110
A	1	10011
C	1	10100
k	1	10101
M	1	11110
SPACE	5	01

Using the frequency, I grouped the characters in ascending order. I built the tree using a bottom up approach beginning with 2 characters of the lowest frequency and merging them together.

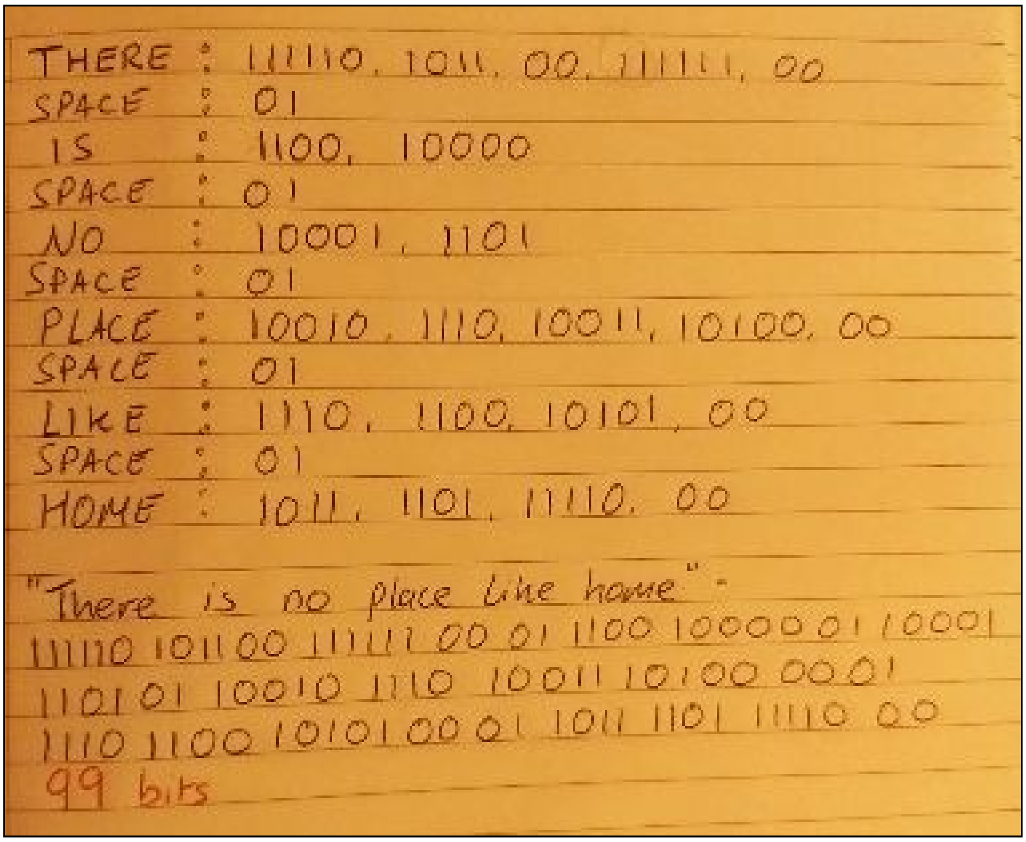
Every time I merged the characters, I gave them a 0 if they were on the left side of the node and a 1 if they were on the right side, by tracing a path from root to leaf node, aggregating the bits along the way. This formed the encoding part of the table.

When I reached the root node, I was able to draw out my full tree that you can see below:



0's were used in the left forks of the tree and 1's were used in the right forks of the tree. The frequency of this tree was 27.

Using the encoding, I was able to generate a codeword table for the phrase, “There is no place like home”. You can see this below:



THERE	:	11110, 1011, 00, 111111, 00
SPACE	:	01
IS	:	1100, 10000
SPACE	:	01
NO	:	10001, 1101
SPACE	:	01
PLACE	:	10010, 1110, 10011, 10100, 00
SPACE	:	01
LIKE	:	1110, 1100, 10101, 00
SPACE	:	01
HOME	:	1011, 1101, 11110, 00

"There is no place like home" -
11110 1011 00 111111 00 01 1100 10000 01 10001
1101 01 10010 1110 10011 10100 00 01
1110 1100 10101 00 01 1011 1101 11110 00
99 bits

The compressed bit string for this phrase took up 99 bits.

Task 2: Coding the Huffman Algorithm in Java
Class path in the github repository is /src/Assignment/Huffman.java

Navigate to the correct directory locally.
Compilation: javac Huffman.java
Execution *Compression:* java Huffman compress inputFile.type outputFile.type
Decompression: java Huffman decompress inputFile.type outputFile.type

Compilation of BinaryDump to get size: javac BinaryDump.java
Execution: java BinaryDump 0 < fileName

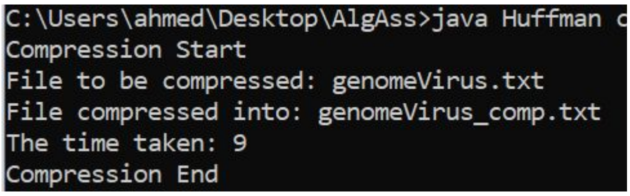
Files used within Huffman: BinaryIn.java; BinaryOut.java; MinPQ.java; StdOut.java

Task 3: Compression Analysis

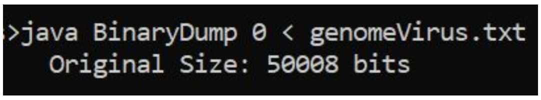
Input File	Output File	Original Size	Compressed Size	Compression Ratio	Time (milliseconds)
genomeVirus.txt	genomeVirus_comp.txt	50008 bits	12576 bits	25.15%	9
medTale.txt	medTale_comp.txt	45808 bits	24616 bits	53.74%	15
mobydick.txt	mobydick_comp.txt	9708952 bits	5505424 bits	56.70%	203
q32x48.bin	q32x48_comp.bin	1536 bits	816 bits	53.13%	6
mine.txt	mine_comp.txt	10808 bits	6288 bits	58.18%	7

All compressions used this command: *java Huffman compress input.type output.type*

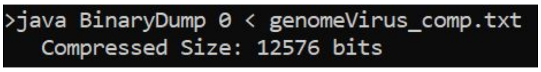
genomeVirus.txt



```
C:\Users\ahmed\Desktop\AlgAss>java Huffman c
Compression Start
File to be compressed: genomeVirus.txt
File compressed into: genomeVirus_comp.txt
The time taken: 9
Compression End
```

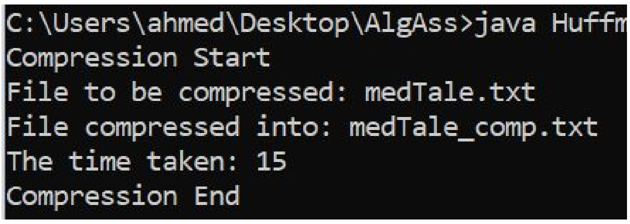


```
>java BinaryDump 0 < genomeVirus.txt
  Original Size: 50008 bits
```

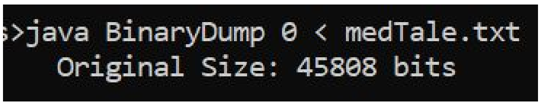


```
>java BinaryDump 0 < genomeVirus_comp.txt
  Compressed Size: 12576 bits
```

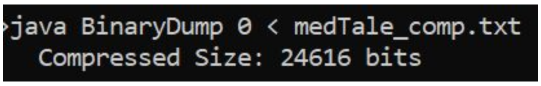
medTale.txt



```
C:\Users\ahmed\Desktop\AlgAss>java Huffm
Compression Start
File to be compressed: medTale.txt
File compressed into: medTale_comp.txt
The time taken: 15
Compression End
```

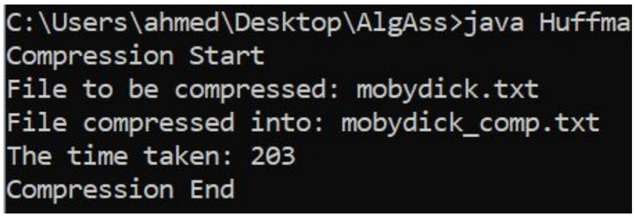


```
>java BinaryDump 0 < medTale.txt
  Original Size: 45808 bits
```

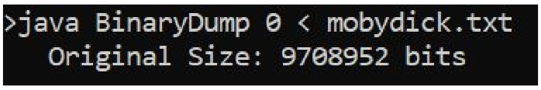


```
java BinaryDump 0 < medTale_comp.txt
  Compressed Size: 24616 bits
```

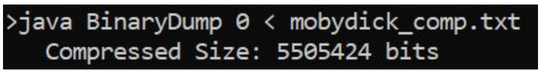
mobydick.txt



```
C:\Users\ahmed\Desktop\AlgAss>java Huffma
Compression Start
File to be compressed: mobydick.txt
File compressed into: mobydick_comp.txt
The time taken: 203
Compression End
```

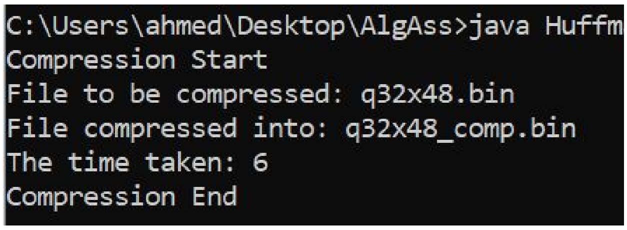


```
>java BinaryDump 0 < mobydick.txt
  Original Size: 9708952 bits
```

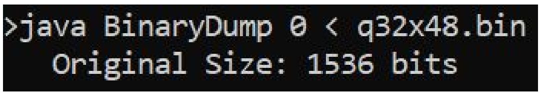


```
>java BinaryDump 0 < mobydick_comp.txt
  Compressed Size: 5505424 bits
```

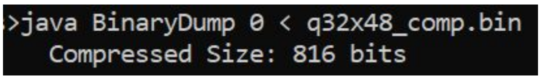
q32x48.bin



```
C:\Users\ahmed\Desktop\AlgAss>java Huffm
Compression Start
File to be compressed: q32x48.bin
File compressed into: q32x48_comp.bin
The time taken: 6
Compression End
```

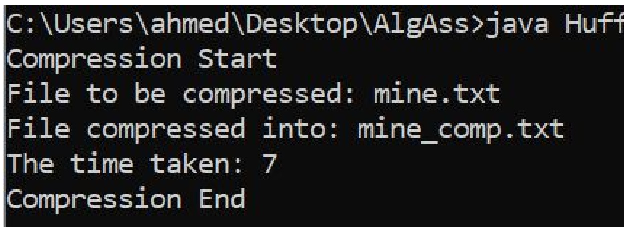


```
>java BinaryDump 0 < q32x48.bin
  Original Size: 1536 bits
```

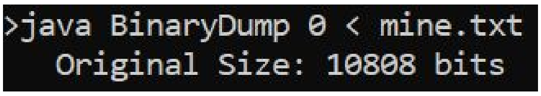


```
>java BinaryDump 0 < q32x48_comp.bin
  Compressed Size: 816 bits
```

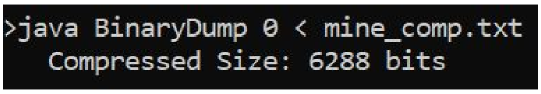
mine.txt



```
C:\Users\ahmed\Desktop\AlgAss>java Huff
Compression Start
File to be compressed: mine.txt
File compressed into: mine_comp.txt
The time taken: 7
Compression End
```



```
>java BinaryDump 0 < mine.txt
  Original Size: 10808 bits
```



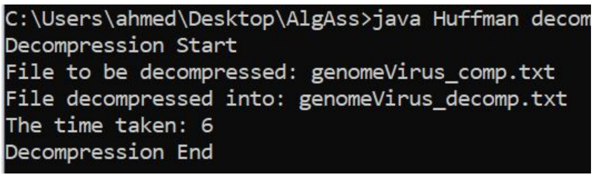
```
>java BinaryDump 0 < mine_comp.txt
  Compressed Size: 6288 bits
```

Decompression Analysis

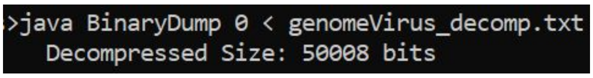
Input File	Output File	Decompressed Size	Time (milliseconds)
genomeVirus_comp.txt	genomeVirus_decomp.txt	50008 bits	6
medTale_comp.txt	medTale_decomp.txt	45808 bits	6
mobydick_comp.txt	mobydick_decomp.txt	9708952 bits	90
q32x48_comp.bin	q32x48_decomp.bin	1536 bits	3
mine_comp.txt	mine_decomp.txt	10808 bits	6

All decompressions used this command: *java Huffman decompress input.type output.type*

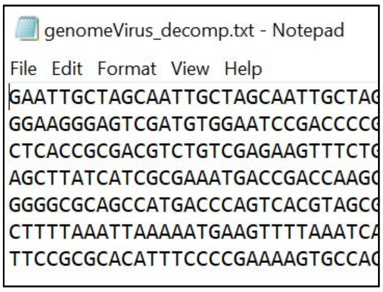
genomeVirus_comp.txt



```
C:\Users\ahmed\Desktop\AlgAss>java Huffman decom
Decompression Start
File to be decompressed: genomeVirus_comp.txt
File decompressed into: genomeVirus_decomp.txt
The time taken: 6
Decompression End
```

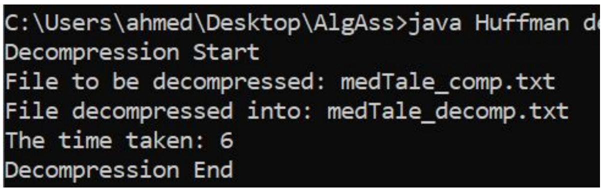


```
>java BinaryDump 0 < genomeVirus_decomp.txt
   Decompressed Size: 50008 bits
```

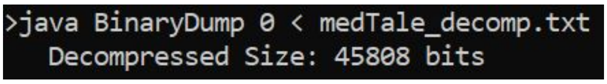


```
GAATTGCTAGCAATTGCTAGCAATTGCTAG
GGAAGGGAGTCGATGTGGAATCCGACCCCG
CTCACCGCGACGTCTGTCGAGAAGTTTCTG
AGCTTATCATCGCGAAATGACCGACCAAGG
GGGGCGCAGCCATGACCCAGTCACGTAGCG
CTTTTAAATTAAAAATGAAGTTTTAAATCA
TTCCGCGCACATTTCCCCGAAAAGTGCCAC
```

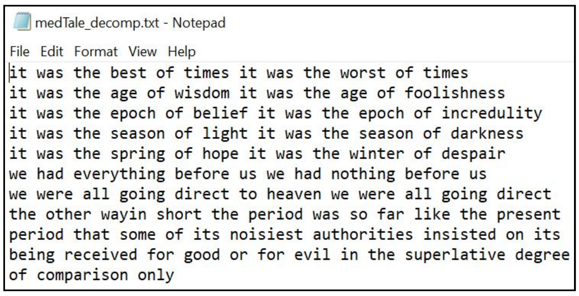
medTale_comp.txt



```
C:\Users\ahmed\Desktop\AlgAss>java Huffman d
Decompression Start
File to be decompressed: medTale_comp.txt
File decompressed into: medTale_decomp.txt
The time taken: 6
Decompression End
```

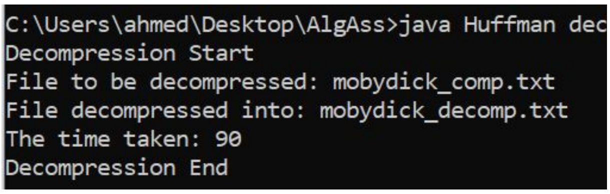


```
>java BinaryDump 0 < medTale_decomp.txt
   Decompressed Size: 45808 bits
```

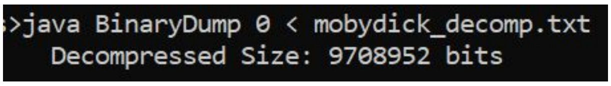


```
it was the best of times it was the worst of times
it was the age of wisdom it was the age of foolishness
it was the epoch of belief it was the epoch of incredulity
it was the season of light it was the season of darkness
it was the spring of hope it was the winter of despair
we had everything before us we had nothing before us
we were all going direct to heaven we were all going direct
the other wayin short the period was so far like the present
period that some of its noisiest authorities insisted on its
being received for good or for evil in the superlative degree
of comparison only
```

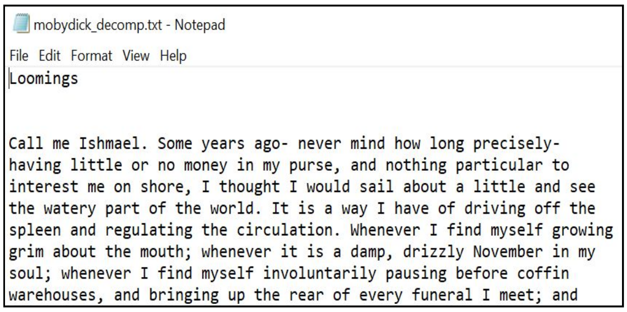
mobydick_comp.txt



```
C:\Users\ahmed\Desktop\AlgAss>java Huffman dec
Decompression Start
File to be decompressed: mobydick_comp.txt
File decompressed into: mobydick_decomp.txt
The time taken: 90
Decompression End
```



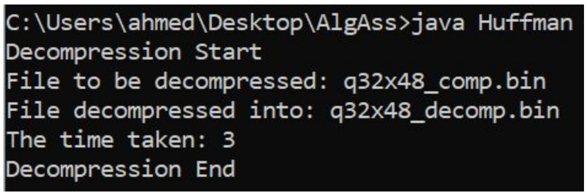
```
>java BinaryDump 0 < mobydick_decomp.txt
   Decompressed Size: 9708952 bits
```



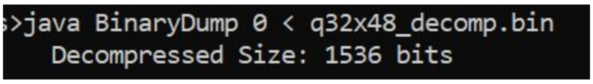
```
Loomings


Call me Ishmael. Some years ago- never mind how long precisely-
having little or no money in my purse, and nothing particular to
interest me on shore, I thought I would sail about a little and see
the watery part of the world. It is a way I have of driving off the
spleen and regulating the circulation. Whenever I find myself growing
grim about the mouth; whenever it is a damp, drizzly November in my
soul; whenever I find myself involuntarily pausing before coffin
warehouses, and bringing up the rear of every funeral I meet; and
```

q32x48_comp.bin

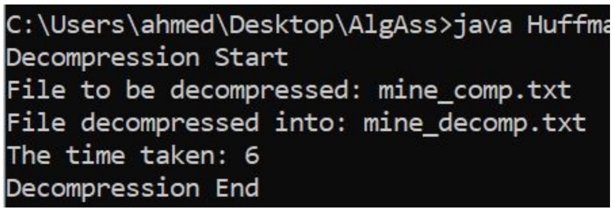


```
C:\Users\ahmed\Desktop\AlgAss>java Huffman
Decompression Start
File to be decompressed: q32x48_comp.bin
File decompressed into: q32x48_decomp.bin
The time taken: 3
Decompression End
```

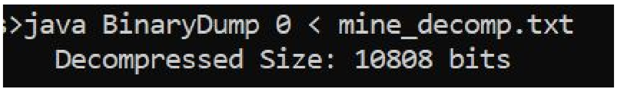


```
>java BinaryDump 0 < q32x48_decomp.bin
   Decompressed Size: 1536 bits
```

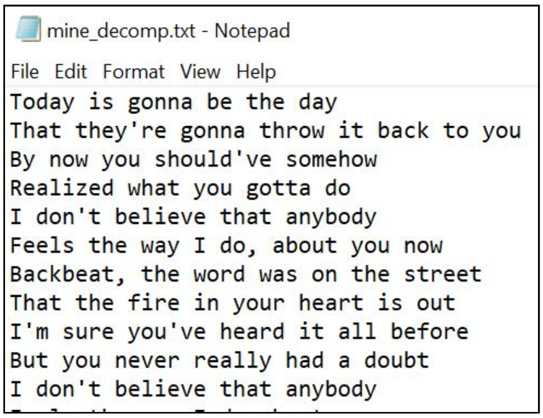
mine_comp.txt



```
C:\Users\ahmed\Desktop\AlgAss>java Huffma
Decompression Start
File to be decompressed: mine_comp.txt
File decompressed into: mine_decomp.txt
The time taken: 6
Decompression End
```



```
>java BinaryDump 0 < mine_decomp.txt
   Decompressed Size: 10808 bits
```



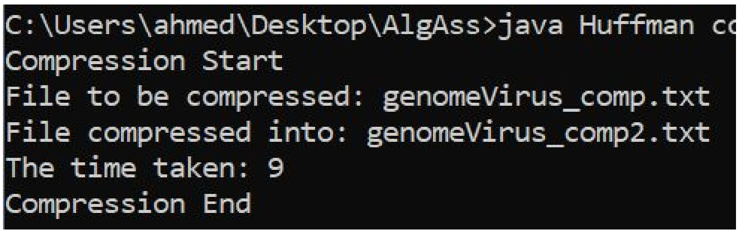
```
Today is gonna be the day
That they're gonna throw it back to you
By now you should've somehow
Realized what you gotta do
I don't believe that anybody
Feels the way I do, about you now
Backbeat, the word was on the street
That the fire in your heart is out
I'm sure you've heard it all before
But you never really had a doubt
I don't believe that anybody
```

Assess the results above:

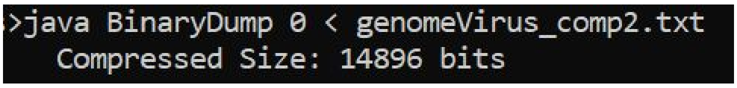
- It is clear that the larger the size of the file the longer compression and decompression take.
- Huffman is clearly an optimum compression algorithm as the maximum compression ratio is 58.18% which is excellent compared to other algorithms such as RunLength encoding which averages a ratio in the 70s.
- Type of file doesn't matter, the algorithm is still efficient.
- Decompression is significantly quicker than compression.
- We can also see that when we decompress a compressed file it results in the exact same original file with no loss as Huffman encoding is a *lossless compression algorithm*.
- Compression and decompression times for a file are proportional.
- The compression ratio depends on how random the file is and doesn't have to do with the size. For example, genomeVirus.txt (50008 bits) got compressed to (12576 bits) a 25.15% compression ratio which is excellent, while medTale.txt (similarly sized, 45808 bits) got compressed to (24616 bits) with a compression ratio of 53.74%. The advantage that genomeVirus has is the alphabet is less and is less random than medTale.

Double Compressing

Try compressing the already compressed genomeVirus_comp



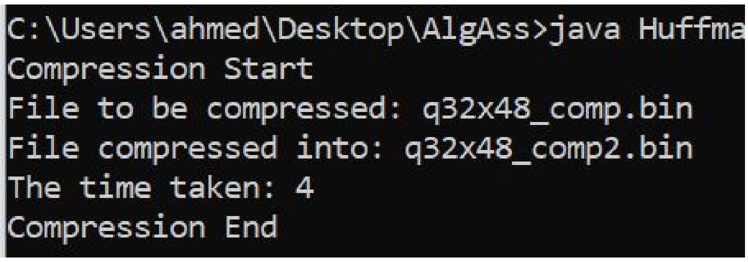
```
C:\Users\ahmed\Desktop\AlgAss>java Huffman c
Compression Start
File to be compressed: genomeVirus_comp.txt
File compressed into: genomeVirus_comp2.txt
The time taken: 9
Compression End
```



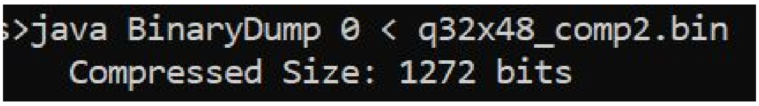
```
>java BinaryDump 0 < genomeVirus_comp2.txt
  Compressed Size: 14896 bits
```

Number of compressions	Size
genomeVirus.txt *(Original - No compressions)*	50008 bits
genomeVirus_comp.txt *(Compressed Once)*	12576 bits
genomeVirus_comp2.txt *(Compressed Twice)*	14896 bits

Try compressing the already compressed q32x48_comp



```
C:\Users\ahmed\Desktop\AlgAss>java Huffma
Compression Start
File to be compressed: q32x48_comp.bin
File compressed into: q32x48_comp2.bin
The time taken: 4
Compression End
```



```
>java BinaryDump 0 < q32x48_comp2.bin
  Compressed Size: 1272 bits
```

Number of compressions	Size
q32x48.bin *(Original - No compressions)*	1536 bits
q32x48_comp.bin *(Compressed Once)*	816 bits
q32x48_comp.bin *(Compressed Twice)*	1272 bits

What happens?

Clearly the second compression is having a negative effect and results in an increase in size. It almost brought back q32x48.bin to the original size almost doubling its size, while in the case of genomeVirus.txt it also resulted in a negative effect with an increase in size but slight.

Why?

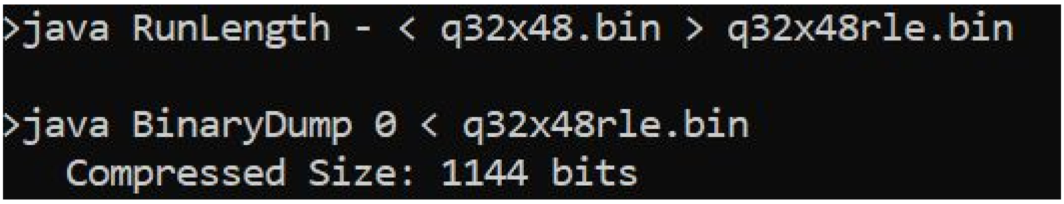
This happens because Huffman coding tries to optimize the alphabet by choosing a representation for each symbol, resulting in a prefix code that expresses the most common symbols using shorter strings of bits than are used for less common

symbols. This way, most of the structure and redundancy have been squeezed out, and what's left looks pretty much like randomness.
No compression algorithm, including Huffman, can effectively compress a random file. Therefore, trying to re-compress a compressed file won't shorten it significantly, and might well lengthen it sometimes. Therefore the optimal number of times to compress a file is usually **one**.

The encoding might end up increasing the number of needed bits as there is no fixed sequence. It's random. For example in practical 9, run length encoding is multiplying the text “ABRACADABRA!” in size instead of compressing it because it is adding an encoding to it when there are no consecutive repetitions of letters.

Alternative:
A way to get a better compressed result that doesn’t risk increasing the size is Double Huffman Coding. Double Huffman Coding is a technique that works on Huffman coding and after getting a codeword for the symbol it is compressed on the basis of its binary no. 0 and 1. This will give a better result than Huffman Coding.

<u>RunLength.java comparison</u>



```
>java RunLength - < q32x48.bin > q32x48rle.bin

>java BinaryDump 0 < q32x48rle.bin
  Compressed Size: 1144 bits
```

Algorithm	New Size	Compression Ratio
RunLength	1144 bits	1144/1536 = 74.48%
Huffman	816 bits	816/1536 = 53.13%

We can clearly see that Huffman performs way better than RunLength compressing the size by an extra 21%.

The reason for this is that RunLength encoding doesn’t have a table in which it stores encoding for certain sequences/symbols unlike Huffman. Therefore overall Huffman yields a better compression. Run length can be extremely inefficient if there aren’t many consecutive identical symbols or if the identical symbols sequence is short (2 or less). On the other hand, Huffman code is nearly optimal as it reduces the number of unused codewords from the terminals of the code tree, it gives a short average code word length and it relates the probability of a source word to the length of its code word. A code that assigns longer codewords to symbols that occur more frequently cannot be optimum. algorithm. Huffman is an example of a greedy algorithm since it builds up the solution piece by piece. Hence, Huffman algorithm is more efficient than RunLength.

Extra: Using RLE followed by Huffman may result in a better result.

Notes:

- All original, compressed and decompressed files have been pushed in the same package. Compressed files' names end with "_comp", while decompressed end with "_decomp".
- Compression Ratio = $\frac{Compressed\ Size\ in\ bits}{Original\ Size\ in\ bits}$
  I rounded it up to two decimal places and changed it to a % for readability
  The higher the percentage the worse the compression.
  Compression ratio can be calculated in other ways but I picked this as it was used in practicals and in the book.
- Avoided redundant comments in code and kept comments to minimal to ensure excellent visual display yet enough information to understand the code.
- As specified in the question document, I used the tool Binary Dump to calculate the bit size.
- If you are trying to compress/decompress an empty file, an error will be thrown. (Empty Stream).